Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors

Marius Grannæs, Magnus Jahre and Lasse Natvig
Norwegian University of Science and Technology
HiPEAC European Network of Excellence
(grannas—jahre—lasse)@idi.ntnu.no

Abstract—The pressure on off-chip memory increases significantly as more cores compete for the same resources. A CMP deals with the memory wall by exploiting thread level parallelism (TLP), shifting the focus from reducing overall memory latency to memory throughput. This extends to the memory controller where the 3D structure of modern DRAM is exploited to increase throughput.

Traditionally, prefetching reduces latency by fetching data before it is needed. In this paper we explore how prefetching can be used to increase memory throughput. We present our own low-cost open-page prefetch scheduler that exploits the 3D structure of modern DRAM when issuing prefetches. We show that because of the complex structure of modern DRAM, prefetches can be made cheaper than ordinary reads, thus making prefetching beneficial even when prefetcher accuracy is low. As a result, prefetching with good coverage is more important than high accuracy. By exploiting this observation our low-cost open page scheme increases performance and QoS. Furthermore, we explore how prefetches should be scheduled in a state of the art memory controller by examining sequential, scheduled region, CZone/Delta Correlation and reference prediction table prefetchers.

I. INTRODUCTION

Chip Multiprocessors have been introduced by virtually all makers of high performance processors. CMPs shifts the focus away from the traditional uniprocessor paradigm, where low latency and instruction-level parallelism (ILP) is important to a paradigm where throughput and thread-level parallelism (TLP) dominates.

This shift is reflected in the memory subsystem as well, where the memory controllers have traditionally been used to reduce system latency. However, as more cores are added to a chip, off-chip bandwidth are shared across cores, thus increasing the pressure on this resource and lowering locality in the memory access stream. Thus, memory controllers have been designed to optimize for maximum throughput, at the expense of increasing worst-case latency.

This increase in throughput has been made possible by exploiting the 3D structure of modern DRAM [1]. DRAM is organized in several banks. Each bank is organized as a matrix of rows and columns of DRAM cells as shown in figure 1. In a normal read operation, a bank and a row is first selected for activation. The charges from this row of capacitors are then amplified by sense-amplifiers in the DRAM module and stored in a large latch. Each such row is commonly referred to as a page. A page is normally about 1KB to 4KB large, whereas a cacheline is typically 64-256B large. Thus, a page will typically hold several consecutive cachelines. The portion of the page that was requested is then transferred over the data-bus. When the page is no longer needed, the memory controller instructs the DRAM module to write the latch contents back into the DRAM cells, preserving the contents of the page. This is referred to as closing the page.

In terms of latency, opening and closing a page is expensive, while getting data out of the latches and over the data-bus is comparatively cheap. In addition, there is a minimum allowed time between opening and closing a page (the minimum activate-to-precharge latency). Thus a single read is slow, but reading the next cache block is relatively cheap as the page is already open and the data is in the latch. This property is exploited by the First Ready, First Come, First Served (FR-FCFS) memory controller proposed by Rixner et al. [2]. This type of memory controller allows accesses that uses an already open page to be scheduled even if the request is not the oldest.

Traditionally, prefetching has been used to decrease latency for a single operation by speculatively bringing data into the cache before it is needed. In this paper we exploit the 3D structure of modern DRAM to demonstrate how prefetching can be used to increase off-chip bandwidth utilization. Because there is a lower cost associated with fetching data that resides in an open page, we prefetch this data, provided that our confidence that the data will be useful is high enough. In addition, we show that prefetching can be effective at relatively low accuracy, due to the low cost of piggybacking prefetches compared to single reads. Finally,
we present our low cost open page prefetching scheduling heuristic which exploits this observation.

II. PREVIOUS WORK

A. Prefetching

Previously, Wei-Fen et al. [3] have examined how prefetches can be scheduled in a uniprocessor context with Rambus DRAM. They used a dedicated prefetch queue with a LIFO insertion policy with a scheduled region prefetcher. In addition, Cantin et al. [4] exploited open pages to increase the performance of their stealth prefetcher.

There exists a multitude of different prefetching schemes. The simplest is the sequential prefetcher [5], which simply fetches the next block whenever a block is referenced. However, more complex types exist as well, such as the CZone/Delta Correlation (C/DC) prefetcher proposed by Nesbit et al. [6, 7]. C/DC divides memory into CZones and analyses patterns contained in the reference stream by using a Global History Buffer (GHB) to store recent misses to the cache. Lin et al. [8] introduced scheduled region prefetching (SRP) which issues prefetches to blocks spatially near the addresses of recent demand misses when the memory channel is idle. Other types, such as the Reference Prediction Table Prefetcher (RPT) proposed by Chen and Baer [9] examines the pattern generated by a load instruction with a state machine. Somogyi et al. proposed Spatial Memory Streaming (SMS) [10]. SMS uses code-correlation to predict spatial access patterns.

B. Memory Controllers

Memory access scheduling is the process of reordering memory requests to improve memory bus utilization. Rixner et al. [2] showed that significant speed-ups are possible when memory request reordering is applied to stream processors. In addition, Shao et al. [11] proposed burst scheduling in which multiple read and write requests to the same DRAM page are issued together to achieve high bus utilization. Finally, Zhu et al. [12] showed that it is beneficial to divide the memory requests into smaller parts, and give priority to the words responsible for a processor stall in a multi-channel DRAM system.

CMPs, processors with SMT support and conventional shared-memory multiprocessors also benefit from memory access scheduling. Zhu et al. [13] showed that DRAM throughput could be increased in an SMT processor by using ROB and IQ occupancy status to prioritize requests. Furthermore, Hur et al. [14] use a history-based arbiter to adapt the DRAM port and rank schedule to the application’s mix of reads and writes for the dual-core Power5 processor. In addition, Natarajan et al. [15] showed that a significant performance improvement is available by exploiting memory controller features in a conventional, shared-memory multiprocessor.

In CMPs, the memory bus is shared between all processing cores and a number of researchers have looked into how this can be accomplished in a fair way [16, 17, 18, 19]. In general, bandwidth is divided among threads according to their priorities at the same time as requests are scheduled in a way that improves DRAM throughput.

III. PREFETCH SCHEDULING

A prefetching heuristic can be characterized by using two distinct metrics: Accuracy is a measure of how many of the issued prefetches have actually been useful to the processor [20], while coverage measures how many of the potential prefetches have been issued.

Because prefetching is a speculative technique, there are two potential sources for performance degradation. Firstly, prefetching consumes additional bandwidth as some data transferred over the memory bus is not used. Secondly, it can pollute the cache, by displacing data that is still needed.

The FR-FCFS memory scheduler [2] is a high throughput memory scheduler. It exploits the 3D structure of modern DRAM by allowing requests that would access an already open page to bypass the normal FCFS queue. FR-FCFS prioritizes memory requests in the following manner: 1) Ready operations (operations that access open pages), 2) CAS (column selection) over RAS (row selection) commands, and 3) Oldest request first. In addition, reads have a higher priority than writes.

There are two basic ways to introduce prefetching into the FR-FCFS memory controller. The simplest approach is to insert prefetch requests into the read queue, as shown in figure 2a. A more sophisticated approach introduced by Lin Wei-Fen et al. [3] is to use a dedicated queue for prefetches as shown in figure 2b. In this approach, prefetches are prioritized after writebacks, so the priority rule becomes: Prioritize read operations over writeback operations over prefetch operations.

IV. LOW COST OPEN PAGE PREFETCHING

After a demand read to DRAM is serviced, the page that the demand read resided in is still open, and in most cases cannot be closed due to the minimum activate to precharge latency. Other DRAM banks can still be utilized. If a prefetch or read is issued to this open page, there is little latency as the data requested is already in the latch. In this paper we refer to this as piggybacking.

By allowing prefetches to piggyback on regular read requests, the cost of prefetching is effectively reduced. In the dedicated prefetch queue approach, prefetches are only issued if they can piggyback on another request, or if the bus is idle. Suppose a processor requires data at locations X_1 and X_2 that are located on the same page at times T_1 and T_2. There are two separate outcomes: If T_1 and T_2 are sufficiently close, both requests will be in the memory controller at the same time, and request 2 can piggyback on request 1. Thus the page only needs to be opened once. If the two requests are sufficiently separated in time, the two requests cannot be piggybacked on each other, thus forcing the page to be opened twice. This reduces overall throughput. In the second case, prefetching X_2 can increase performance by both reducing latency and increase memory throughput. However, because prefetching is a speculative technique, its
prediction for what data is needed in the future might be wrong. Thus, there is a break-even point where the benefit of prefetching is balanced against the cost of prefetching.

To test this assumption we have conducted experiments on 4 different prefetching heuristics (Sequential, SRP, C/DC and RPT) with 10 different prefetching configurations (each) on 40 different workloads. We measured the accuracy of the prefetcher and the IPC improvement (versus a configuration with no prefetching). Our results are shown in figure 3.

In this graph it is clear that most of the points fall into 2 quadrants. One where accuracy is below 38% and performance is decreased, while another where accuracy is above 38% and performance is increased.

Our prefetch scheduler exploits this observation by measuring prefetch accuracy at runtime. If the accuracy falls below a threshold (in our experiments 38%) then prefetches are no longer piggybacked on open pages and only issued if the bus is idle.

We use an accuracy estimator similar to the one used by Sriniath et al. [21]. When a prefetch is issued, a counter is increased (indicating the number of prefetches issued) and a prefetched-bit is set in the corresponding cache line. This bit is already present when using sequential prefetching, and thus causes no additional overhead. The first time a cache line with this bit set is referenced by the program, the bit is cleared and another counter (indicating the number of successful prefetches) is increased. By sampling the successful prefetch counter every time the 10 bit issued counter wraps, we get an estimate of the prefetchers accuracy.

V. METHODOLOGY

We used the system call emulation mode of the cycle-accurate M5 simulator [22] to evaluate our scheme. The processor architecture parameters for the simulated 4-core CMP are shown in table I, and table II contains the baseline memory system parameters. We have extended M5 with a crossbar interconnect, a detailed DDR2 memory bus and DRAM model, a FR-FCFS memory controller and prefetching.

Our DDR2-implementation [23] models separate RAS, CAS and precharge commands. In addition, we model pipelining of requests, independent banks, burst mode transfers and bus contention. The FR-FCFS memory controller has a 128 entry read-queue, 64 entry writeback queue and a 128 entry prefetch queue. As the conventional method of issuing prefetches has no separate prefetch queue, the read queue has been increased to 256 entries to make comparison more fair in terms of area. Unless otherwise noted, we use 4KB CZones in scheduled region prefetching, 256KB C Zones, a 1024-entry global history buffer and a 16-entry reference prediction table.

The SPEC CPU2000 benchmark suite [24] is used to create 40 multiprogrammed workloads consisting of 4 SPEC benchmarks each as shown in table III. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the random selection process was that each SPEC benchmark had to be represented in at least one workload. To avoid unrealistic interference when more than a single instance of a benchmark is part of a workload, the benchmarks are fast-forwarded a random number of clock cycles between 1 and 1.1 billion. Then, detailed simulation is carried out for 100 million clock cycles measured from the clock cycle the last core finished fast forwarding. As our metric of throughput we have used the average IPC of all 4 cores. In most cases, performance is measured as the relative increase in speed compared to the no prefetching case.

VI. RESULTS

A. Scheduled Region Prefetching

In figure 4 we show the relative performance of each of the prefetch scheduling policies. In this experiment we use a scheduled region prefetcher (SRP) with 4KB regions. The conventional and dedicated prefetch queue options give an average of 14.4% increase in performance versus the no prefetching case, while the average increase for our scheme is 17.1%. In addition, prefetching causes performance degradation in 9 out of the 40 cases. Our prefech scheduling policy reduces the performance penalty on 6 of these workloads. However, a lot of information is lost in averages. For instance, the performance increased on workload 1 is only 1% in other schemes, while our method increases performance by 15%. Similar results can be seen in workload 6, 7, 23, 25, 27, 28, 32 and 38.
Fig. 3: IPC improvement as a function of accuracy

Fig. 4: Speedup in IPC relative to no prefetching using a FR-FCFS memory controller.
TABLE I: Processor Core Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Cores</td>
<td>4</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>3.2 GHz</td>
</tr>
<tr>
<td>Reorder Buffer</td>
<td>128 entries</td>
</tr>
<tr>
<td>Store Buffer</td>
<td>32 entries</td>
</tr>
<tr>
<td>Instruction Queue</td>
<td>64 instructions</td>
</tr>
<tr>
<td>Instruction Fetch Queue</td>
<td>32 entries</td>
</tr>
<tr>
<td>Load/Store Queue</td>
<td>32 instructions</td>
</tr>
<tr>
<td>Issue Width</td>
<td>8 instructions/cycle</td>
</tr>
<tr>
<td>Functional units</td>
<td>4 Integer ALUs, 2 Integer Multiply/Divide, 4 FP ALUs, 2 FP Multiply/Divide</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>Hybrid, 2048 local history registers, 4-way 2048 entry BTB</td>
</tr>
</tbody>
</table>

TABLE II: Memory System Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 Data Cache</td>
<td>64 KB, 8-way set associative, 64B blocks, 3 cycles latency</td>
</tr>
<tr>
<td>Level 1 Instruction Cache</td>
<td>64 KB, 8-way set associative, 64B blocks, 1 cycle latency</td>
</tr>
<tr>
<td>Level 2 Unified Shared Cache</td>
<td>4 MB, 16-way set associative, 64B blocks, 14 cycles latency, 8 MSHRs per bank, 4 banks</td>
</tr>
<tr>
<td>L1 to L2 Interconnection Network</td>
<td>Crossbar topology, 9 cycles latency, 64B wide transmission channel</td>
</tr>
<tr>
<td>DDR2 memory</td>
<td>400 Mhz Clock, 8 banks, 1KB pagesize, 4-4-4-12 timing, dual channel in lock-step</td>
</tr>
</tbody>
</table>

Fig. 5: Average speedup in IPC relative to no prefetching.

B. Importance of Coverage

In figure 5 we show the average relative performance increase by using other types of prefetchers, including Scheduled Region Prefetching, CZone/Delta Correlation and Reference Prediction Tables. Both C/DC and RPT prefetching have high accuracy. Because the prefetching accuracy is higher than the threshold in almost all workloads, our method degrades into the dedicated prefetch queue. In turn, the performance of our prefetch scheduling scheme is almost equal to the dedicated prefetch queue scheme. However, this graph shows another interesting property. Scheduled Region Prefetching, which has a comparatively low accuracy, outperforms both of the more complex prefetcher heuristics. This is due to it having a much higher prefetch coverage. It provides more prefetches with acceptable accuracy, thus increasing performance.

C. Insertion policy

In our scheme and the dedicated prefetch queue scheme there is a separate queue for handling prefetches. There are multiple possibilities on how to insert new prefetches into the queue. If the prefetch queue is full, then there are two possibilities, either discard the prefetch or insert the prefetch and evict the oldest prefetch. In figure 6 we show the performance of FIFO and LIFO policies with and without evictions. From this graph it is clear that evicting old data is beneficial, as well as using a LIFO policy. Evicting old prefetches is useful, because newer prefetches are based on newer demand reads, thus increasing both the accuracy and the probability that it can be piggybacked. The LIFO policy ensures that the newest prefetches are given priority over old ones. As shown in the graph, for both techniques, evicting old data is preferable, while a LIFO policy gives marginally better results over FIFO.

D. Treshold parameter

In figure 7 we show the average speedup as a function of the required accuracy (threshold). In effect, setting the threshold to 0% makes the low cost open page prefetcher a dedicated queue prefetcher. Both RPT and C/DC prefetching have a very high accuracy, so the threshold doesn’t affect performance until it becomes too high, effectively disabling prefetching, and in turn degrades performance. In addition, the peak for both sequential and scheduled region prefetching is relatively low (around 20-30%). This further supports the observation that coverage is more important as long as accuracy is acceptable.
VII. Discussion

Our results show that it is more important to have good prefetching coverage, while having acceptable accuracy. This is due to the relatively lower cost of piggybacked prefetches compared to isolated demand reads. Normally prefetch heuristics have been optimized for maximizing accuracy, so that the impact on bandwidth is as low as possible. This is due to the assumption that the cost of a single prefetch is about the same as a demand read. By carefully scheduling prefetches so that they are piggybacked on normal demand reads, this assumption no longer holds. We have demonstrated that a simpler, high coverage prefetcher outperforms more sophisticated high accuracy prefetchers in a bandwidth-constrained, 4-core chip multiprocessor system.

In our prefetch scheduling heuristic, we have used an accuracy estimator to control when prefetches should be issued. Other researchers have used such an estimator to control the aggressiveness of the prefetcher [21]. Such a technique can be used in conjunction with our scheduler. By using a feedback directed prefetcher, coverage can be increased while keeping accuracy at an acceptable level, thus providing higher performance.

Our simulator does not include a power model. However, our scheme piggybacks prefetches on demand reads. If a prefetch is successful then a later read is not needed, thus reducing the number of pages opened and closed, which in turn reduces power consumption in the DRAM module. Prefetching invariably increases bus traffic as some data transferred is not needed. Our scheme reduces the amount of useless traffic compared to other schemes by filtering out prefetches with low accuracy, thereby saving power.

VIII. Conclusion

In this paper we have shown that by carefully scheduling prefetches so that they piggyback on ordinary demand reads, performance can be increased. This is done by exploiting the 3D structure of modern DRAM, where opening and closing pages is an expensive operation. As it becomes more important to issue prefetches that can be piggybacked on ordinary demand reads, emphasis shifts from high accuracy to high coverage with acceptable accuracy.

We have demonstrated our own prefetch scheme on a state of the art memory controller that exploits these findings. Our prefetch policy outperforms traditional scheduling policies in terms of performance, quality of service and power consumption.

REFERENCES


E. Quality of Service

We have measured the maximum slowdown for any thread compared to the case where no prefetching is performed on each workload to get an indicator of the quality of service. Figure 8 shows the maximum performance degradation as a function of the number of workloads included. This graph shows three important properties. Firstly, 25% of the workloads experience no performance degradation on any thread when doing prefetching. Secondly, our scheme gives consistently higher quality of service. Using the other scheme 33% of the workloads show a thread getting a performance degradation of above 10%. In our scheme only 20% of the workloads show a thread getting more than 10% performance degradation. Finally, the maximum degradation for any thread for our scheme is only 36%, while the maximum for the dedicated prefetch queue approach is 49%.
TABLE III: Multiprogrammed Workloads

<table>
<thead>
<tr>
<th>ID</th>
<th>Benchmarks</th>
<th>ID</th>
<th>Benchmarks</th>
<th>ID</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ammp, mgrid, perl, apsi, parser</td>
<td>11</td>
<td>vpr, twolf, applu, eon</td>
<td>21</td>
<td>perl, apsi, lucas, equake</td>
</tr>
<tr>
<td>2</td>
<td>lucas, gcc, mcf, twolf</td>
<td>12</td>
<td>galgel, crafty, mgrid, swim</td>
<td>22</td>
<td>vpr, crafty, vpr, mcf</td>
</tr>
<tr>
<td>3</td>
<td>eon, eon, mesa, facerec</td>
<td>13</td>
<td>twolf, fma3d, galgel, vpr</td>
<td>23</td>
<td>gzip, equake, mgrid, mesa</td>
</tr>
<tr>
<td>4</td>
<td>vortex1, ammp, equake, galgel</td>
<td>14</td>
<td>bzip, vpr, bzip, equake</td>
<td>24</td>
<td>facerec, applu, fma3d, lucas</td>
</tr>
<tr>
<td>5</td>
<td>gcc, galgel, apsi, crafty</td>
<td>15</td>
<td>mcf, crafty, vpr, swim</td>
<td>25</td>
<td>gap, applu, parser, facerec</td>
</tr>
<tr>
<td>6</td>
<td>applu, equake, art, facerec</td>
<td>16</td>
<td>mcf, wupwise, mesa, mesa</td>
<td>26</td>
<td>mcf, apsi, twolf, ammp</td>
</tr>
<tr>
<td>7</td>
<td>applu, gap, gcc, parser</td>
<td>17</td>
<td>applu, parser, apsi, perl, mgrid</td>
<td>27</td>
<td>mcf, equn, mesa, vertex1</td>
</tr>
<tr>
<td>8</td>
<td>gap, swim, twolf, mesa</td>
<td>18</td>
<td>mgrid, perl, bzip, gzip, mgrid</td>
<td>28</td>
<td>art, fma3d, swim, parser</td>
</tr>
<tr>
<td>9</td>
<td>sixtrack, fma3d, apsi, vertex1</td>
<td>19</td>
<td>mcf, sixtrack, gcc, apsi</td>
<td>29</td>
<td>apsi, gcc, vertex1, twolf</td>
</tr>
<tr>
<td>10</td>
<td>ammp, bzip, equake, parser</td>
<td>20</td>
<td>ammp, gcc, art, mesa</td>
<td>30</td>
<td>mgrid, gzip, apsi, equake</td>
</tr>
</tbody>
</table>